

Ordenação por partes (radixSort)

Na última aula vimos o **countingSort**,

- cuja ideia central é contar o número de predecessores de cada chave
 - para posicionar corretamente os elementos no vetor ordenado.
- Ele é **muito eficiente** para ordenar conjuntos de elementos
 - cujas chaves são inteiros pequenos.
- Por inteiros pequenos, queremos dizer valores inteiros entre 0 e $R-1$
 - sendo R limitado pelo número de elementos do conjunto.

Quiz1: Para perceber a limitação do countingSort, considere

- um vetor com **1 milhão** de elementos cujas chaves são inteiros de 32 bits.
 - Qual a eficiência do countingSort neste caso?

Uma alternativa para ordenar conjuntos cujas chaves são grandes

- é dividi-las em pedaços menores e ordenar por etapas.
 - Essa é a ideia central dos métodos de ordenação **radixSort**.
- Chamamos cada um dos pedaços da chave de dígito
 - e seu tamanho deriva da base (radix) utilizada.
- Assim, os dígitos não pertencem necessariamente ao conjunto $\{0, \dots, 9\}$.

Os métodos **radixSort** também são chamados de ordenação digital, por ordenar

- as chaves dígito-a-dígito, ou as strings caractere-a-caractere.
- Eles variam de acordo com a ordem em que consideramos os dígitos,
 - i.e., se vamos dos mais significativos para os menos,
 - ou dos menos para os mais significativos.

⇒ **LSD radixSort** $\left\{ \begin{array}{l} \text{Least} \\ \text{Significative} \\ \text{Digit} \end{array} \right.$

Este método é interessante para ordenar um vetor $V[0..n-1]$ de chaves,

- sendo todas do mesmo comprimento.
- Se as chaves são strings de caracteres,
 - estamos falando do problema de colocá-las em ordem lexicográfica.
- Ordenamos indo do dígito menos significativo até o mais significativo,
 - i.e., percorremos cada chave da direita para a esquerda.

chave $\left[\begin{array}{|c|c|c|c|} \hline 11 & 22 & 33 & 44 \\ \hline \end{array} \right]$

- Em cada etapa ordenamos todo o conjunto, considerando apenas um dígito
 - e usando um método de ordenação estável.
 - **Quiz2:** Por que ser estável é essencial?
- De preferência, usamos o countingSort,
 - por ser estável e muito rápido com chaves pequenas (dígitos).

Primeiro veremos uma versão do LSD radixSort

- que trabalha com vetores de caracteres (strings) de mesmo tamanho,
 - sendo cada string uma chave.

Exemplo:

Original	3° Dígito	2° Dígito	1° Dígito
1 2 3	1 2 3	1 2 3	1 2 3
KNG -	KDA	KDA	FFU
FFU -	RFD	KDV	KDA
KDV -	KNG	RFD	KDV
RFD -	FFU	FFU	KNG
KDA -	KDV	KNG	RFD

Observe que na última coluna todas as chaves estão ordenadas.

- Para entender o motivo, considere a penúltima coluna e observe que
 - se remover dela todos os elementos que não começam com K
 - os elementos restantes já estão em ordem.
- De fato, no início da i -ésima etapa deste método
 - todas as chaves estão ordenadas
 - com relação apenas aos dígitos já considerados,
 - i.e., com índices maiores que i
- Como a ordenação utilizada em cada etapa é estável,
 - esta propriedade invariante se mantém
 - e propaga para mais um dígito a cada nova etapa.

No algoritmo a seguir,

- W é o comprimento, em dígitos, de cada chave (string),
- R é o universo de valores que cada dígito pode assumir.
 - Note que, $R \leq 256$ já que cada dígito corresponde a um byte.

$1 \text{ byte} = 8 \text{ bits}$
 $2^8 = 256$
 possibilidades

Códigos:

```
typedef unsigned char byte;
```

```
// Rearranja em ordem Lexicográfica um vetor v[0 .. n - 1]
// de strings. Cada v[i] é uma string v[i][0 .. W - 1]
// cujos elementos pertencem ao conjunto 0 .. R - 1.
```

```
void ordenacaoDigital(byte *v[], int n, int W) {
    int *ocorr_pred, digito, valor, i, R = 256;
    byte **aux;
    ocorr_pred = malloc((R + 1) * sizeof(int));
    aux = malloc(n * sizeof(byte *));
```

```

⇒ for (digito = W - 1; digito >= 0; digito--) {
    for (valor = 0; valor <= R; valor++) ocorrencia_pred[valor] = 0;
    for (i = 0; i < n; i++) {
        valor = v[i][digito];
        ocorrencia_pred[valor + 1] += 1;
    } // ocorrencia_pred[valor] é o # de ocorrências de valor - 1
    for (valor = 1; valor <= R; valor++)
        ocorrencia_pred[valor] += ocorrencia_pred[valor - 1];
    // ocorrencia_pred[valor] é o # de ocor. dos pred. de valor
    for (i = 0; i < n; i++) {
        valor = v[i][digito];
        aux[ocorrencia_pred[valor]] = v[i];
        ocorrencia_pred[valor]++; // atualiza número de predecessores
    }
    // aux[0..n-1] está em ordem crescente considerando
    // apenas os dígitos entre digito e W - 1
    for (i = 0; i < n; i++) v[i] = aux[i];
}

free(ocorrencia_pred); free(aux);
}

```

basically counting Sort

$O(R)$

$O(n)$

$O(R)$

detalhes na aula de counting Sort

$O(n)$

$O(n)$

Invariante e corretude: No início de cada iteração do laço externo

- as strings estão ordenadas com relação
 - aos subvetores $v[i][\text{digito}+1 \dots W-1]$, $\forall i = 0, \dots, n-1$

Eficiência de tempo: Uma observação importante para analisar o método radixSort

- é que nenhum dígito é verificado mais de uma vez.
- Assim, radixSort é linear no número total de dígitos, i.e.,
 - $O((n+R)W)$ sendo n o número de chaves,
 - R o número de valores que cada dígito pode assumir, e
 - W o número de dígitos em cada chave.
- Este método é melhor que ordenações $O(n \lg n)$ quando $R = O(n)$ e
 - $W = O(\lg n)$, sendo $O(\lg n)$ assintoticamente menor que $\lg n$

Quiz3: Considere ordenar 1 milhão de elementos com chaves de 5 dígitos.

- Compare $\lg$ do número de elementos com o número de dígitos das chaves
 - para decidir se o LSD radix é preferível ao quickSort, por exemplo.

Eficiência de espaço: Memória adicional da ordem de $(n+R)$, i.e., $O(n+R)$

- por conta dos vetores auxiliares do countingSort.

Estabilidade: É estável, pois como o método que ordena cada dígito é estável,

- em nenhuma etapa elementos com a mesma chave serão invertidos,
 - já que eles coincidem em todos os dígitos.

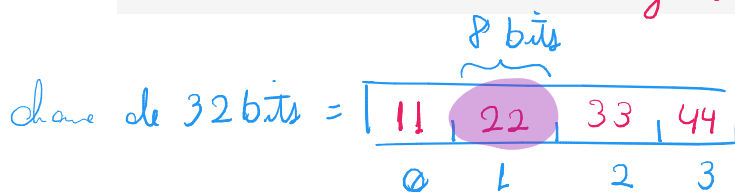
Agora veremos uma versão semelhante do **LSD radixSort**

- que trabalha com vetores de inteiros, sendo cada inteiro uma chave.
- Esta versão manipula os bits das chaves para obter cada dígito.

Códigos:

```
const int bitsPalavra = 32;
const int bitsDigito = 8;
const int digitosPalavra = bitsPalavra / bitsDigito;
const int Base = 1 << bitsDigito; // Base = 2^bitsDigito

int pegaDigito(int chave, int digito) {
    return (int)((chave >>
        (bitsDigito * (digitosPalavra - 1 - digito))) & (Base - 1));
}
```



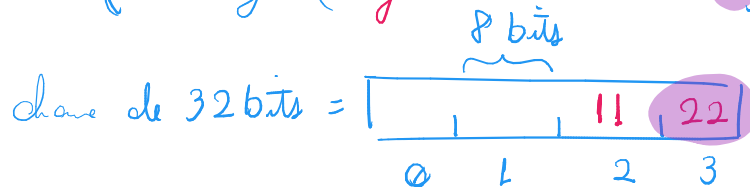
$$\text{digitos Palavra} = 32/8 = 4$$

$$\text{digito} = d = 1$$

$$(\text{digitos Palavra} - 1 - d) = 4 - 1 - 1 = 2$$

$$(\text{bits Digito}(\text{digitos Palavra} - 1 - d)) = 8 \cdot 2 = 16$$

$$(chave >> (\text{bits Digito}(\text{digitos Palavra} - 1 - d))) = chave >> 16$$

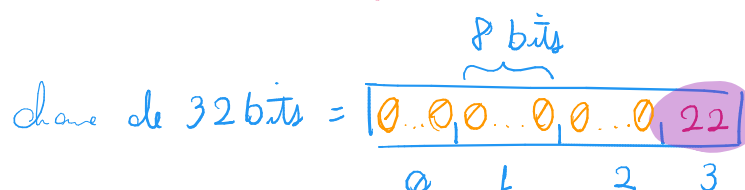


$$\text{Base} = 1 \ll \text{bits Digito} = 1 \ll 8$$

$$\text{Base} = 0 \dots 0010 \dots 0$$

$$\text{Base} - 1 = 0 \dots 001 \dots 1$$

$$(chave >> (\text{bits Digito}(\text{digitos Palavra} - 1 - d))) \& (\text{Base} - 1)$$



```

int pegaDigito2(int chave, int digito) {
    return (int)(chave /  $\rightarrow$  divisão inteira
    exp2(bitsDigito * (digitosPalavra - 1 - digito))) % Base;
}

```

de dígitos à direita do dígito desejado

o resto da divisão inteira

```

void LSDradixSort(int v[], int n) {
    int digito, valor, i;
    int *ocorr_pred, *aux;
    - occurr_pred = malloc((Base + 1) * sizeof(int));
    - aux = malloc(n * sizeof(int));

     $\Rightarrow$  for (digito = [digitosPalavra - 1]; digito >= 0; digito  $\rightarrow$ ) {
        ( for (valor = 0; valor <= Base; valor++)
            occurr_pred[valor] = 0; ) =  $O(\text{Base})$ 
        for (i = 0; i < n; i++) {
            valor = pegaDigito(v[i], digito);
            occurr_pred[valor + 1] += 1;
        }

         $\Rightarrow$  // agora occurr_pred[valor] é o # de ocorrências de valor - 1
        ( for (valor = 1; valor <= Base; valor++)
            occurr_pred[valor] += occurr_pred[valor - 1]; ) =  $O(\text{Base})$ 
        // agora occurr_pred[valor] é o número de
        // ocorrências dos predecessores de valor.
        // Logo, a sequência de elementos iguais a valor
        // deve começar no índice occurr_pred[valor].
        for (i = 0; i < n; ++i) {
            // note a diferença entre o valor analisado e copiado  $\Leftarrow$ 
            valor = pegaDigito(v[i], digito);
            aux[occurr_pred[valor]] = v[i];  $\leftarrow$ 
            occurr_pred[valor]++; // atualiza o # de predecessores
        }
        // aux[0..n-1] está em ordem crescente considerando
        // apenas os dígitos entre digito .. digitosPalavra - 1
        for (i = 0; i < n; i++)
            v[i] = aux[i];
    }

    free(occurr_pred); -
    free(aux); -
}

```

Eficiência de tempo:

- Uma observação importante para analisar o método radixSort é que
 - cada dígito é verificado um número constante de vezes.
- Assim, radixSort é linear no número de dígitos, i.e.,
 - $O((n + Base) \cdot \text{dígitosPalavra})$, sendo n o número de chaves;
 - $Base$ o número de valores que cada dígito pode assumir;
 - Note que, sendo bitsDígito o número de bits de cada dígito,
 - $Base = 2^{\text{bitsDígito}}$
 - dígitosPalavra o número de dígitos em cada chave.
 - Note que, sendo bitsPalavra o número de bits por chave,
 - $\text{dígitosPalavra} = \text{bitsPalavra} / \text{bitsDígito}$

Se os dígitos forem pequenos e as chaves grandes

- a complexidade de tempo pode superar $n \lg n$ assintoticamente.
- Já se os dígitos forem grandes em relação às chaves
 - a eficiência tende a $O(n)$
- Por exemplo, tome $n = 1 \text{ bilhão} = 10^9$
 - Se $\text{bitsDígito} = 1$ ($Base = 2^1 = 2$) e $\text{bitsPalavra} = 64$
 - temos $\text{dígitosPalavra} = 64 / 1 = 64$
 - que é maior que $\lg 10^9 \approx \lg 2^{10 \cdot 3} \approx 30$
 - Se $\text{bitsDígito} = 16$ ($Base = 2^{16} = 65536$) e $\text{bitsPalavra} = 32$
 - temos $\text{dígitosPalavra} = 32 / 16 = 2$
 - que é muito menor que $\lg 10^9 \approx 30$

Quiz4: Para perceber a vantagem do radixSort, considere um vetor

- com 1 milhão de elementos cujas chaves são inteiros de 32 bits.
 - Qual a eficiência do radixSort com dígito de 8 bits?

Eficiência de espaço: Memória adicional da ordem de $(n + Base)$

- i.e., $O(n + Base)$
- por conta dos vetores auxiliares do countingSort.

Observação: Notem que, quando o tamanho dos dígitos cresce,

- a Base cresce exponencialmente,
- o que significa maior consumo de tempo e de espaço.

Estabilidade: É estável, pois como o método que ordena cada dígito é estável,

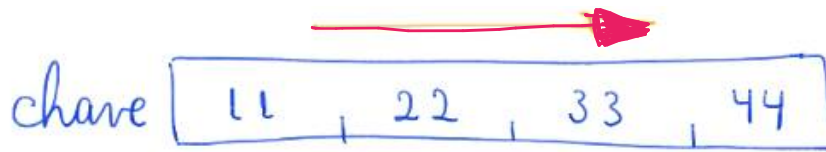
- em nenhuma etapa elementos com a mesma chave serão invertidos,
 - já que eles coincidem em todos os dígitos.

MSD radixSort

Most Significant Digit

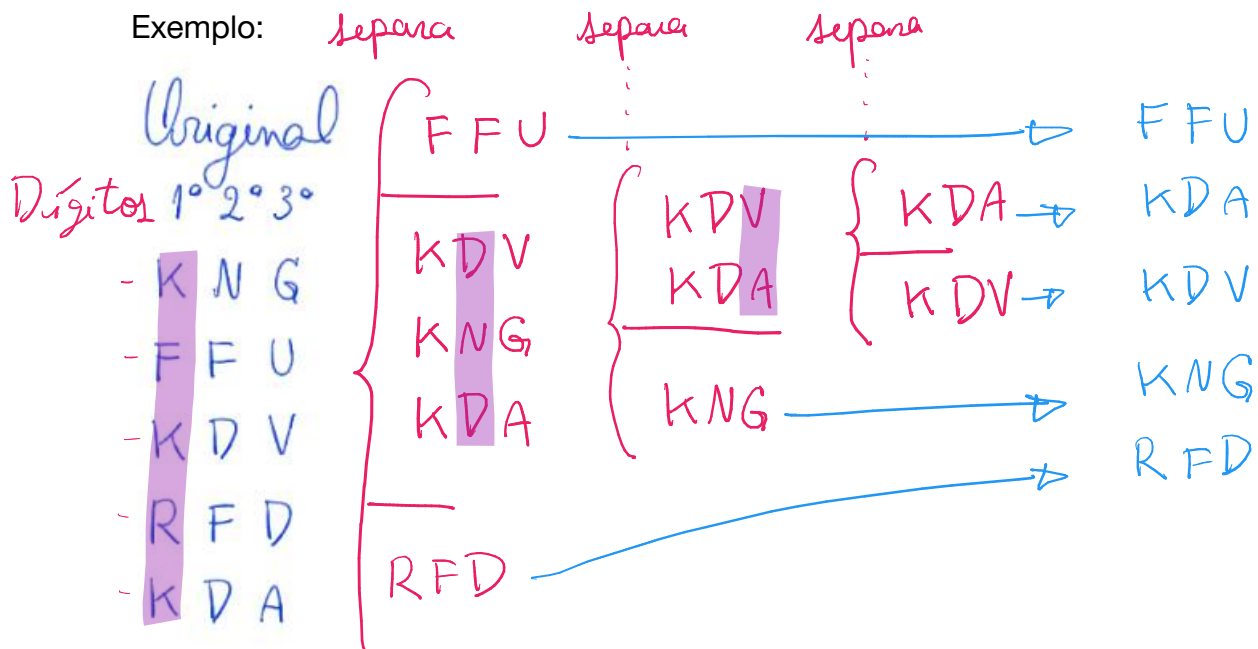
Neste método vamos ordenar o conjunto

- indo do dígito mais significativo até o menos significativo,
 - i.e., percorremos cada chave da esquerda para a direita.



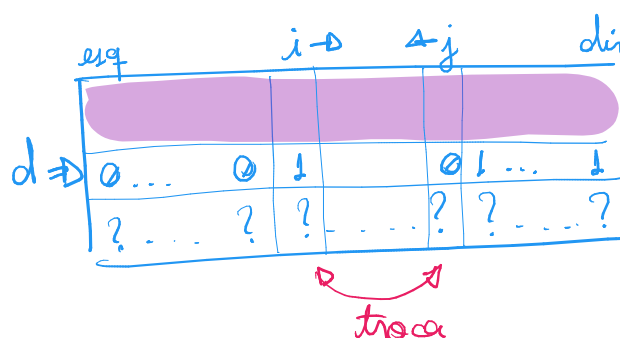
- Em cada etapa podemos usar uma variante do algoritmo da separação,
 - que estudamos junto do quickSort.
- Esta variante só considera o dígito corrente
 - e divide o conjunto em Base subconjuntos,
 - sendo Base o número de possibilidades de valor de um dígito.
- Este método pode lidar com chaves de comprimentos variados,
 - e nem sempre precisa avaliar todos os dígitos de todas as chaves,
 - o que pode melhorar sua eficiência.

Exemplo:



Vamos focar em um caso particular, e mais simples, do MSD radixSort em que

- o dígito tem tamanho 1, e Base = $2^1 = 2$, chamado de binary quickSort.
- Segue um exemplo de separação que coloca elementos com
 - dígito 0 à esquerda e dígito 1 à direita.



A seguir temos uma implementação do binary quickSort.

Códigos:

```
const int bitsPalavra = 32;
const int bitsDigito = 1; → o código a seguir só funciona c/ esse valor
const int digitosPalavra = bitsPalavra / bitsDigito;
const int Base = 1 << bitsDigito; // Base = 2^bitsDigito
```

// *esq indica a 1a posição, dir a ultima, digito é o digito corrente*

```
void quickSortBin(int v[], int esq, int dir, int digito) {
```

```
    int i = esq, j = dir;
```

// caso base quando subvetor é pequeno ou acabaram os dígitos

caso base →

```
if (dir <= esq || digito > digitosPalavra) return;
```

// separa as chaves do subvetor de acordo com o digito corrente

```
while (j > i) {
```

```
    - while (pegaDigito(v[i], digito) == 0 && i < j) i++;
```

```
    - while (pegaDigito(v[j], digito) == 1 && j > i) j--;
```

```
    troca(&v[i], &v[j]);
```

```
}
```

// ajusta j para que v[esq .. j-1] tenha chaves com digito 0

```
if (pegaDigito(v[esq], digito) == 0) j++; ↔
```

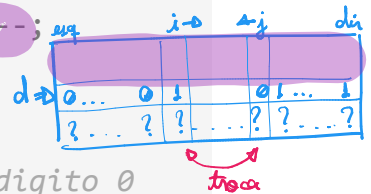
```
quickSortBin(v, esq, j - 1, digito + 1);
```

```
quickSortBin(v, j, dir, digito + 1);
```

1º p.

2º p.

```
- void MSDradixSort(int v[], int n) {quickSortBin(v, 0, n - 1, 0);}
```



Eficiência de tempo: Uma observação importante para analisar os método radixSort

- é que nenhum dígito é verificado mais que um # constante de vezes.
- Assim, no pior caso radixSort é linear no número de dígitos, i.e.,
 - $O(n \cdot \text{digitosPalavra}) = O(n \cdot \text{bitsPalavra})$ no caso do binary quickSort.

Eficiência de espaço: Memória adicional da ordem de digitosPalavra = bitsPalavra

- por conta da profundidade máxima da pilha de recursão.

Estabilidade: Não é estável, por conta das trocas da separação.

Quiz5: Note que, essa implementação trata chaves com comprimentos variados,

- desde que não existam chaves repetidas. Por que?
- Dica: analise as condições do caso base.

Quiz6: Como seria uma versão deste algoritmo com bitsDigito > 1?

- Dica: solução para separação inspirada
 - na contagem de predecessores do countingSort.
- Por conta disso, um termo R = Base surge na eficiência do algoritmo.